Inteligencia Artificial 2017-2018

Práctica 2: Búsqueda

GRUPO: 2213

CELIA SAN GREGORIO MORENO

ÁLVARO MARTÍNEZ MORALES

Índice

MODELIZACIÓN DEL PROBLEMA 3
Ejercicio 1: Evaluación del valor de la heurística 3
Código 3
Comentario 3
Ejercicio 2: Operadores navigate-worm-hole y navigate-white-hole 4
Código 4
Comentario 5
Ejercicio 3A: Test para determinar si se ha alcanzado el objetivo 6
Código 6
Comentario 7
Ejercicio 3B: Predicado para determinar la igualdad entre estados de búsqueda 8
Código 8
Comentario 8

FORMALIZACIÓN DEL PROBLEMA 9
Ejercicio 4: Representación LISP del problema 9
Código 9
Comentario 9
Ejercicio 5: Expandir nodo 10
Código 10
Comentario 11
Ejercicio 6: Gestión de nodos 12
Código 12
Comentario 13
BÚSQUEDAS 14
Ejercicio 7: Definir estrategia para la búsqueda A* 14
Código 14
Comentario 14
Ejercicio 8: Función de búsqueda 15
Código 15
Comentario 15
Ejercicio 9: Ver el camino seguido y la secuencia de acciones 18

Código..... 18

Comentario..... 19

Ejercicio 10: Otras estrategias de búsqueda 20

Código..... 20

Comentario..... 20

REFLEXIÓN 22

Ejercicio 11: Ejercicios de reflexión 22

11.1. ¿Por qué se ha realizado este diseño para resolver el problema de búsqueda? ¿Qué ventajas aporta? ¿Por qué se han utilizado funciones lambda para especificar el test objetivo, la heurística y los operadores del problema?..... 22

11.2. Sabiendo que en cada nodo de búsqueda hay un campo "parent", que proporciona una referencia al nodo a partir del cual se ha generado el actual ¿es eficiente el uso de memoria?..... 22

11.3 ¿Cuál es la complejidad espacial del algoritmo implementado?..... 22

11.4 ¿Cuál es la complejidad temporal del algoritmo?..... 23

11.5 Indicad qué partes del código se modificarían para limitar el número de veces que se puede utilizar la acción "navegar por agujeros de gusano" (bidireccionales)..... 23

MODELIZACIÓN DEL PROBLEMA

Ejercicio 1:

Evaluación del valor de la heurística

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 1 -- Evaluation of the heuristic
;;
;; Returns the value of the heuristics for a given state
;;
;;  Input:
;;    state: the current state (vis. the planet we are on)
;;    sensors: a sensor list, that is a list of pairs
;;               (state cost)
;;            where the first element is the name of a state and the second
;;            a number estimating the cost to reach the goal
;;
;;  Returns:
;;    The cost (a number) or NIL if the state is not in the sensor list
;;
(defun f-h-galaxy (state sensors)
  (second (assoc state sensors)))

;;
;; END: Exercise 1 -- Evaluation of the heuristic
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Comentario

El ejercicio nos pedía calcular el valor de la heurística en un estado dado.

Dado que este cálculo ya viene *'hardcodeado'* en la estructura de la implementación base de la práctica, la única codificación necesaria fue la extracción de ese dato de la estructura de los sensores.

Ejercicio 2:

Operadores *navigate-worm-hole* y *navigate-white-hole*

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 2 -- Navigation operators
;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Crea una lista de tripletes con state como planeta de origen
;; y cada uno de los planetas en el agujero blanco o de gusano
;; a los que se puede acceder desde state.
;;
;;  Input:
;;    state: el estado actual (el planeta donde estamos)
;;    hole-map: lista de tripletes correspondiente a los grafos
;;              de la galaxia (en este caso, agujeros blancos
;;              o de gusano).
;;
;;  Returns:
;;    Lista de tripletes de tipo (<state> <planeta-destino> <coste>).
(defun make-colindant-list (state hole-map)
  ;Si hemos llegado al final de la lista
  ;asociativa, la función termina.
  (if (null hole-map)
      nil
    ;Si no, comprueba si el planeta de origen (state)
    ;coincide con el planeta de origen del primer triplete.
    (if (equal state (first (first hole-map)))
        ;Si coincide, crea una lista de tripletes.
        (cons (first hole-map)
              (make-colindant-list state (rest hole-map)))
      ;Sino, avanza en la lista asociativa hole-map.
      (make-colindant-list state (rest hole-map)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Operador genérico que devuelve una lista de acciones que se
;; pueden hacer a partir del estado state, sobre un
;; grafo cualquiera, con posibildad de exclusión.
;;
;;  Input:
;;    state: estado de búsqueda que representa al planeta de origen.
;;    hole-map: lista de tripletes correspondiente al grafo cualquiera.
;;    forbidden: planetas que no permitir como destino. De no haberlos, debe ser nil
;;    action-name: nombre que asignar a la acción.
;;
;;  Returns:
;;    Lista de acciones de la acción definida del planeta de origen al de destino.
(defun navigate (state hole-map forbidden action-name)
  ;Genera una lista de acciones con los resultados de
  ;la función 'make-colindant-list' no presentes en forbidden.
  (mapcan #'(lambda (dest)
              (if (member (second dest) forbidden)
                  nil
                (list (make-action
                       :name action-name
                       :origin state
                       :final (second dest)
                       :cost (third dest)))))
          (make-colindant-list state hole-map)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;
;; Operador que devuelve una lista de acciones que se
;; pueden hacer a partir del estado state, sobre un
;; grafo con agujeros blancos.
;;
;;  Input:
;;    state: estado de búsqueda que representa al planeta de origen.
;;    white-holes: lista de tripletes correspondiente al grafo de
;;                 agujeros blancos de la galaxia.
;;
;;  Returns:
;;    Lista de acciones del planeta de origen al de destino, a
;;    través de los agujeros blancos.
(defun navigate-white-hole (state white-holes)
  (navigate state white-holes nil 'navigate-white-hole))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Operador que devuelve una lista de acciones que se
;; pueden hacer a partir del estado state, sobre un
;; grafo con agujeros de gusano.
;;
;;  Input:
;;    state: estado de búsqueda que representa al planeta de origen.
;;    white-holes: lista de tripletes correspondiente al grafo de
;;                 agujeros de gusano de la galaxia.
;;
;;  Returns:
;;    Lista de acciones del planeta de origen al de destino, a
;;    través de los agujeros de gusano.
(defun navigate-worm-hole (state worm-holes planets-forbidden)
  (navigate state worm-holes planets-forbidden 'navigate-worm-hole))

;;
;; END: Exercise 2 -- Navigation operators
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Comentario

El ejercicio consistía en, dado un estado, elaborar un listado con las acciones que pudieran tener lugar desde el mismo en base a los distintos mapas estelares a nuestra disposición (en este caso, el mapa de *worm holes* y *white holes*. Esto, claro, implica la exclusión de los *'planetas prohibidos'* en el caso de los *worm holes* como posibles destinos.

Esto se ha implementado, fundamentalmente, mediante dos funciones principales y una interfaz para cada tipo de mapa.
La *primera función principal* es **make-colindant-list**, que devuelve todos los estados colindantes a un estado dado en base al mapa indicado.
La *segunda función principal* es **navigate**, que, dado un estado, un mapa, una posible declaración de nodos prohibidos y el nombre de una acción, produce, mediante la primera función, una lista de acciones con cada nodo colindante en el que se excluyen los planetas prohibidos, de haberlos.

Por último, las interfaces son una capa de transparencia sobre la función principal *navigate*, que permite, aparte de especificar la acción para cada tipo de mapa, sólo pedir la información requerida para cada uno de los dos tipos de mapas.

Ejercicio 3A:

Test para determinar si se ha alcanzado el objetivo

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 3A -- Goal test
;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Comprueba si el nodo pasado como argumento es un estado objetivo.
;;
;;  Input:
;;    nodo: nodo que representa un estado de búsqueda (el planeta actual).
;;    planets-destination: lista de nombres de los planetas destino.
;;    planets-mandatory: lista de nombres de los planetas obligatorios.
;;
;;  Returns:
;;    T si el nodo es un estado objetivo, NIL si no.
(defun f-goal-test-galaxy (node planets-destination planets-mandatory)
  ;Si el nodo está entre la lista de planetas destino,
  ;comprueba que los nodos antecesores hayan pasado por
  ;los planetas obligatorios.
  (if (member (node-state node) planets-destination)
     ;(f-mandatory-test node planets-mandatory)
     (f-mandatory-test (node-parent node) planets-mandatory) ;Comprueba si los nodos padre
corresponden
    nil))                                   ;a planetas obligatorios visitados.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Devuelve una lista de planetas obligatorios aún no visitados.
;;
;;  Input:
;;    node: nodo que representa un estado de búsqueda (el planeta actual).
;;    planets-mandatory: lista de nombres de los planetas obligatorios.
;;
;;  Returns:
;;    Lista con los nombres de los planetas obligatorios que aún
;;    queden por visitar, o NIL si se han visitado todos.
(defun get-mandatory-planets-not-visited (node planets-mandatory)
  ;Si llegamos al nodo raíz, devolvemos
  ;la lista de planetas que quedan por visitar.
  (if (null node)
     planets-mandatory
    ;Si aún no hemos llegado al nodo raíz, comprueba si el nodo actual
    ;es un planeta obligatorio.
    (if (member (node-state node) planets-mandatory :test #'equal)
       ;Si es un planeta obligatorio, lo elimina de la lista y pasa a comprobar el nodo padre.
       (get-mandatory-planets-not-visited (node-parent node) (remove (node-state node) planets-
mandatory))
     ;Si no es obligatorio, pasa a comprobar el nodo padre directamente.
     (get-mandatory-planets-not-visited (node-parent node) planets-mandatory))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Comprueba si en el camino del nodo raíz al nodo actual
;; se ha pasado por los planetas obligatorios.
;;
;;  Input:
;;    nodo: nodo que representa un estado de búsqueda (el planeta actual).
;;    planets-mandatory: lista de nombres de los planetas obligatorios.
;;
;;  Returns:
```

```
;;   T si se ha pasado por todos los nodos obligatorios, NIL si no.
(defun f-mandatory-test (node planets-mandatory)
  ;Y la lista de planetas obligatorios está vacía,
  ;hemos pasado por todos los planetas obligatorios.
  (if (null (get-mandatory-planets-not-visited node planets-mandatory))
      T
    nil))

;;
;; END: Exercise 3A -- Goal test
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Comentario

En primer lugar, hay que tener en cuenta que la modularización del ejercicio se ha llevado a cabo teniendo en cuenta las funcionalidades del ejercicio 3B.

Así, la funcionalidad del ejercicio se ha dividido en tres funciones:

- **get-mandatory-planets-not-visited**, que recibe un nodo y, mediante sus padres, devuelve una lista con los planetas obligatorios sin visitar (o *nil*, de haberlos visitados todos), implementada mediante la eliminación recursiva de los planetas visitados de la lista de planetas obligatorios.
- **f-mandatory-test**, que, mediante la función previa, indica con *T* si la lista devuelta es *nil* (es decir, que todos los planetas obligatorios han sido visitados) o con *nil* si se ha devuelto una lista no vacía, en cuyo caso no ha visitado todos los planetas objetivo.
  Así, esta función indica si se ha pasado por todos los planetas objetivos de camino al nodo dado o no.
- **f-goal-test-galaxy**, que, mediante la función previa y la comprobación de si el nodo dado es el nodo objetivo, indica con *T* si este supone un estado objetivo o *nil* de no serlo.

Ejercicio 3B:

Predicado para determinar la igualdad entre estados de búsqueda

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 3B -- Node equality
;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Comprueba si dos nodos son iguales mediante estos criterios:
;;
;;  - Mismo estado de búsqueda (nombre de planeta), si no se
;;    especifican planetas obligatorios como parámetro.
;;  - Mismo estado de búsqueda y lista de planetas obligatorios
;;    por visitar, si se especifican planetas obligatorios.
;;
;;  Input:
;;    node-1: nodo que representa un estado de búsqueda (un planeta).
;;    node-2: nodo que representa otro estado de búsqueda.
;;    planets-mandatory: lista de nombres de los planetas obligatorios.
;;
;;  Returns:
;;    T si los nodos son iguales, NIL si no.
(defun f-search-state-equal-galaxy (node-1 node-2 &optional planets-mandatory)
  ;Si alguno de los nodos pasados como parámetro es NIL,
  ;la función termina.
  (if (or (null node-1) (null node-2))
      nil
    (let ((planet-1 (node-state node-1))
          (planet-2 (node-state node-2)))
      ;Si no se han especificado planetas obligatorios,
      ;se comprueba si el nombre de los planetas es igual.
      (if (null planets-mandatory)
          (equal planet-1 planet-2)
        ;En caso contrario, comprueba si el nombre de los planetas
        ;y la lista de planetas por visitar coinciden.
        (let ((planets-not-visited-node-1 (get-mandatory-planets-not-visited (node-parent node-1) planets-
mandatory))
              (planets-not-visited-node-2 (get-mandatory-planets-not-visited (node-parent node-2) planets-
mandatory)))
          (and (equal planet-1 planet-2)
               (equal planets-not-visited-node-1 planets-not-visited-node-2)))))))

;;
;; END: Exercise 3B -- Node equality
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Comentario

El código solicitado por el ejercicio ha de comprobar si dos nodos son iguales de cara a la consecución del objetivo.

Esto se ha llevado a cabo tomando dos consideraciones; que el *state* del nodo sea el mismo y que la lista devuelta por la función **get-mandatory-planets-not-visited** (implementada en el apartado 3A) es la misma. De esta forma, si dos nodos tienen el mismo *state* y lista de planetas obligatorios por visitar, ambos son iguales.

FORMALIZACIÓN DEL PROBLEMA

Ejercicio 4:

Representación LISP del problema

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  BEGIN: Exercise 4 -- Define the galaxy structure
;;
;;
(defparameter *galaxy-M35*
 (make-problem
  :states         *planets*
  :initial-state    *planet-origin*
  :f-goal-test      #'(lambda (node)
                    (f-goal-test-galaxy node *planets-destination*
                                *planets-mandatory*))
  :f-h            #'(lambda (state)
                    (f-h-galaxy state *sensors*))
  :f-search-state-equal #'(lambda (node-1 node-2)
                     (f-search-state-equal-galaxy node-1 node-2))
  :operators        (list #'(lambda (node)
                      (navigate-white-hole (node-state node) *white-holes*))
                     #'(lambda (node)
                      (navigate-worm-hole (node-state node) *worm-holes* *planets-forbidden*)))))

;;
;;  END: Exercise 4 -- Define the galaxy structure
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Comentario

En este ejercicio construimos la galaxia *galaxy-M35* como una estructura de tipo *problem* con los siguientes campos:

- **states**. Los planetas que contiene la galaxia.
- **initial-state.** El planeta de origen, a partir del cual la función de búsqueda comenzará a expandir nodos y determinar el camino óptimo al nodo destino.
- **f-goal-test.** Función lambda que comprueba si un nodo que corresponde a un planeta es el planeta de destino (ver ejercicio 3A).
- **f-h.** Función lambda que calcula el valor de la heurística en un estado dado (ver ejercicio 1).
- **f-search-state-equal.** Función lambda que comprueba si dos nodos son iguales (ver ejercicio 3B).
- **operators.** Lista de funciones lambda que corresponden a los distintos operadores que permiten viajar de un planeta origen a uno o más planetas destino a través de un mapa concreto (agujeros blancos o de gusano; ver ejercicio 2).

Ejercicio 5:
Expandir nodo

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN Exercise 5: Expand node
;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Obtiene la lista de nodos a los que se puede acceder
;; desde el nodo actual, utilizando todos los operadores
;; (agujeros blancos y de gusano).
;;
;;  Input:
;;    node: nodo que representa un estado de búsqueda (el planeta actual).
;;    problem: problema de búsqueda.
;;
;;  Returns:
;;    Lista de nodos directamente accesibles desde el nodo actual,
;;    teniendo en cuenta todos los operadores del problema.
(defun expand-node (node problem)
  (expand-node-aux node (problem-operators problem) problem))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Obtiene una lista de nodos a los que se puede acceder
;; desde el nodo actual, de forma recursiva y teniendo
;; en cuenta todos los operadores del problema.
;;
;;  Input:
;;    node: nodo que representa un estado de búsqueda (el planeta actual).
;;    op-list: lista de operadores del problema (en este caso, agujeros
;;          blancos y de gusano).
;;    problem: problema de búsqueda.
;;
;;  Returns:
;;    Lista de nodos directamente accesibles desde el nodo actual,
;;    teniendo en cuenta todos los operadores del problema.
(defun expand-node-aux (node op-list problem)
  ;Si llega al final de la lista de operadores, termina.
  (if (null op-list)
      nil
    ;Sino, crea una lista con cada uno de los nodos
    ;obtenidos a partir de la información de las acciones desde el nodo actual.
    (append (create-node-list-from-action-list (funcall (first op-list) node) node problem)
          (expand-node-aux node (rest op-list) problem))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Crea una lista de nodos a partir de una lista de acciones.
;;
;;  Input:
;;    a-list: lista de acciones que se pueden hacer desde el nodo actual.
;;    node: nodo que representa el estado de búsqueda actual.
;;    problem: problema de búsqueda.
;;
;;  Returns:
;;    Lista de nodos directamente accesibles desde el nodo actual.
(defun create-node-list-from-action-list (a-list parent-node problem)
  ;Fin de la lista de acciones: termina.
  (if (null a-list)
      nil
    ;Crea una lista de nodos a partir de la información
```

```
  ;de cada acción.
  (cons (let* ((nstate (action-final (first a-list)))
            ;(ng (action-cost (first a-list)))
            (ng (+ (action-cost (first a-list)) (node-g parent-node))) ;Coste desde la raíz hasta el nodo actual.
            (nh (funcall (problem-f-h problem) nstate)))
        (make-node
         :state nstate
         :parent parent-node
         :action (first a-list)
         :depth (+ 1 (node-depth parent-node))
         :g ng
         :h nh
         :f (+ ng nh)))
       (create-node-list-from-action-list (rest a-list) parent-node problem))))

;;
;; END Exercise 5: Expand node
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Comentario

El ejercicio consistía en 'expandir un nodo dado', lo que quiere decir, fundamentalmente, obtener una lista de todos los nodos colindantes en base a los mapas de nodos facilitados para mediante la estructura del problema.

Esto se ha llevado a cabo, básicamente, aplicando recursividad de manera transparente mediante la función **expand-node-aux**, que aplica las extracciones de acciones.

Este último paso se lleva a cabo en cada iteración mediante la función **create-node-list-from-action-list**, que, básicamente, de cada lista de acciones obtenida mediante las funciones *navigate* incluidas en la estructura del problema, crea, de nuevo, recursivamente, nuevos nodos, con el nodo expandido como padre, y sacando toda la información de la estructura de acción facilitada.

Por último, se devuelve esa lista de nodos, resultado de la expansión del nodo dado.

Ejercicio 6:
Gestión de nodos

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  BEGIN Exercise 6 -- Node list management
;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Obtiene una lista de nodos ordenada según el criterio
;; de comparación especificado en la estrategia strategy.
;;
;;  Input:
;;    nodes: lista de nodos sin ordenar.
;;    lst-nodes: lista de nodos ordenada según la función de
;;              comparación de strategy.
;;    strategy: estrategia de búsqueda.
;;
;;  Returns:
;;    Lista de nodos a la que se ha añadido cada nodo de nodes,
;;    todos ellos ordenados según el criterio de strategy.
(defun insert-nodes-strategy (nodes lst-nodes strategy)
  ;Si la lista de nodos está vacía, termina.
  (if (null nodes)
      lst-nodes
    ;Sino, va añadiendo cada nodo de nodes a la lista ordenada
    ;lst-nodes mediante sucesivas llamadas a insert-node-strategy.
    (insert-nodes-strategy (rest nodes)
                           (insert-node-strategy (first nodes)
                                                 lst-nodes
                                                 strategy)
                           strategy)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Inserta un nodo en la lista ordenada de nodos de acuerdo
;; al criterio de comparación indicado por strategy.
;;
;;  Input:
;;    node: nodo que se va a insertar en la lista.
;;    lst-nodes: lista de nodos ordenada según la función de
;;              comparación de strategy.
;;    strategy: estrategia de búsqueda.
;;
;;  Returns:
;;    Lista de nodos a la que se ha añadido el nodo,
;;    ordenada según el criterio de strategy.
(defun insert-node-strategy (node lst-nodes strategy)
  ;Si la lista de nodos ordenada por g está vacía, termina.
  (if (null lst-nodes)
      (list node)
    ;Si la función de comparación de strategy indica que el
    ;parámetro a comparar de node es menor que el primer
    ;elemento de lst-nodes...
    (if (funcall (strategy-node-compare-p strategy)
                 node
                 (first lst-nodes))
        ;Node pasa a ser el primer elemento de la lista ordenada.
        (cons node lst-nodes)
      ;Sino, sigue mirando en qué posición insertar el nodo de acuerdo al orden.
      (cons (first lst-nodes) (insert-node-strategy node (rest lst-nodes) strategy)))))

;;
```

```
;; Función de coste uniforme
;;
(defun node-g-<= (node-1 node-2)
  (<= (node-g node-1)
      (node-g node-2)))

;;
;; Estrategia de coste uniforme
;;
(defparameter *uniform-cost*
  (make-strategy
   :name 'uniform-cost
   :node-compare-p #'node-g-<=))

;;
;;    END: Exercise 6 -- Node list management
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Comentario

El ejercicio consistía en ordenar una lista de nodos en una lista ya ordenada bajo una estrategia dada.

Esto se ha llevado a cabo mediante dos funciones; una principal, **insert-nodes-stategy**, que, recursivamente y hasta acabar la lista de nodos, hace uso de la función auxiliar **insert-node-strategy**, que introduce un solo nodo en la lista ordenada, recorriéndola elemento a elemento y comprobando si ha de introducirse ante alguno de ellos, o uniéndola al final en caso de que no encuentre ningún elemento al que deba preceder en base a la estrategia facilitada.

Así mismo, se nos pedía implementar la estrategia ***uniform-cost*** que, a grandes rasgos, suponía crear la función de comparación **node-g-<=**, la cual comprueba si el primer nodo pasado por argumento tiene una valor de *g* menor o igual al del segundo nodo facilitado por argumento.

BÚSQUEDAS

Ejercicio 7:

Definir estrategia para la búsqueda A*

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 7 -- Definition of the A* strategy
;;
;; A strategy is, basically, a comparison function between nodes to tell
;; us which nodes should be analyzed first. In the A* strategy, the first
;; node to be analyzed is the one with the smallest value of g+h
;;
(defun lower-g+h (node1 node2)
  (<= (node-f node1)
      (node-f node2)))

(defparameter *A-star*
  (make-strategy
   :name 'A-star
   :node-compare-p #'lower-g+h))

;;
;; END: Exercise 7 -- Definition of the A* strategy
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Comentario

Para implementar la estrategia de búsqueda A*, hemos definido una estructura de tipo *strategy* con su nombre (*A-star*) y una función de comparación que evaluará a *t* si el valor de f de node1 es menor o igual que el valor de f de node2. Si no se cumple esta condición, devolverá *nil*.

Ejercicio 8:

Función de búsqueda

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    BEGIN Exercise 8: Search algorithm
;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; A partir de un problema de búsqueda, busca la solución
;; óptima (camino más corto) desde el planeta de origen
;; hasta el planeta de destino, siguiendo una estrategia
;; definida.
;;
;;  Input:
;;    problem: problema de búsqueda.
;;    strategy: estrategia de búsqueda.
;;
;;  Returns:
;;    Camino desde el nodo raíz hasta el nodo objetivo.
(defun graph-search (problem strategy)
  ;Inicializa el nodo raíz de la búsqueda,
  ;la lista abierta y la lista cerrada.
  (let* ((initial-planet (problem-initial-state problem))    ;Nombre del planeta inicial.
        (nh (funcall (problem-f-h problem) initial-planet)));Valor de h del nodo raíz.
    (graph-search-rec
     (list (make-node :state initial-planet       ;Nodo raíz del problema con el planeta inicial.
                      :parent nil
                      :action nil
                      :depth 0
                      :g 0
                      :h nh
                      :f (+ 0 nh)))
     nil problem strategy)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; A partir de un problema de búsqueda, busca la solución
;; óptima (camino más corto), de forma recursiva, desde el planeta
;; de origen hasta el planeta de destino, siguiendo una estrategia
;; definida.
;;
;;  Input:
;;    open-nodes: lista abierta que contiene los nodos generados,
;;              pero no expandidos.
;;    closed-nodes: lista cerrada que contiene los nodos generados
;;               y expandidos previamente.
;;    problem: problema de búsqueda.
;;    strategy: estrategia de búsqueda.
;;
;;  Returns:
;;    Camino desde el nodo raíz hasta el nodo objetivo.
(defun graph-search-rec (open-nodes closed-nodes problem strategy)
  (if (null open-nodes)
    nil
```

```
  (let ((current-node (first open-nodes)))
   ;Comprueba si el nodo a expandir es el objetivo.
   (if (f-goal-test-galaxy current-node *planets-destination* *planets-mandatory*)
      ;Si lo es, lo devuelve como solución.
      current-node
     ;En caso contrario, comprueba si el nodo no está en la lista cerrada o,
     ;si está en ella, si tiene un valor de g inferior al primer nodo de closed-nodes.
     ;(if (or (null (member current-node closed-nodes))
     (if (not-in-closed-nodes current-node closed-nodes)
        ;Expande el nodo actual e inserta los hijos en open-nodes, ordenados de
        ;acuerdo al criterio de comparación de strategy.
        ;También inserta el nodo actual en la lista cerrada closed-nodes.
        (let ((new-open-nodes (insert-nodes-strategy (expand-node current-node problem) open-nodes strategy))
              (new-closed-nodes (append (list current-node) closed-nodes)))
          ;Continúa la búsqueda eliminando el nodo expandido actual
          ;de la lista abierta.
          (graph-search-rec (remove current-node new-open-nodes) new-closed-nodes problem strategy))
       ;Si el nodo a expandir no cumple las condiciones, se elimina directamente
       ;de la lista abierta.
       (graph-search-rec (remove current-node open-nodes) closed-nodes problem strategy))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Función que comprueba si el nodo no está en la lista cerrada,
;; o si está, que su valor de g sea menor que el nodo homólogo.
;;
;;  Input:
;;    node: nodo cuya presencia y g comprobar.
;;    node-list: lista de nodos donde comprobar la presencia y el g.
;;
;;  Returns:
;;    T o nil, según se cumpla la condición o no.
(defun not-in-closed-nodes (node node-list)
  (if (null node-list)
      T
    (if (and (f-search-state-equal-galaxy node (first node-list) *planets-mandatory*)
         (> (node-g node) (node-g (first node-list))))
        nil
      (not-in-closed-nodes node (rest node-list)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Resuelve un problema de búsqueda utilizando la estrategia A*.
;;
;;  Input:
;;    problem: problema de búsqueda.
;;
;;  Returns:
;;    Camino desde el nodo raíz hasta el nodo objetivo.
(defun a-star-search (problem)
  (graph-search problem *A-star*))
```

Comentario

El ejercicio consistía en implementar, en primer lugar, una estrategia de búsqueda en grafo, y a partir de ella construir una estrategia de búsqueda de tipo A*.

Para realizar este ejercicio, se ha dividido su funcionalidad en una serie de métodos:

- **graph-search.** A partir de un problema y una estrategia de búsqueda, llama a la función **graph-search-rec** con una lista abierta compuesta por el nodo raíz (construido a partir del estado de origen), una lista cerrada vacía, el problema y la estrategia. El resultado de graph-search-rec será el resultado final de la búsqueda.
- **graph-search-rec.** Esta función sigue, paso a paso, el algoritmo de búsqueda en grafo de manera recursiva.
  En primer lugar comprueba si la lista abierta está vacía. Si lo está, devuelve *nil*, ya que no hay más nodos que expandir y ninguno de ellos cumple el test de objetivo. Si la lista abierta contiene al menos un nodo, realiza un test de objetivo sobre dicho nodo. En caso de ser un nodo objetivo, devuelve su valor. En caso contrario, comprueba si el nodo no está en la lista cerrada, o si está, si su valor de g es menor que el de su homólogo.
  Si se cumplen estas condiciones, dicho nodo se expande y sus hijos se introducen en la lista abierta de acuerdo a la función de comparación de la estrategia. El nodo se añade también al principio de la lista cerrada, se elimina de la lista abierta y la función sigue expandiendo nodos.
  En cambio, si el nodo a expandir está en la lista cerrada y tiene un valor de g mayor que su homólogo, se elimina directamente de la lista abierta y la función sigue expandiendo nodos de forma recursiva.
- **not-in-closed-nodes.** Función auxiliar utilizada por graph-search-rec que comprueba si un nodo no está en la lista cerrada o, si está, su valor de g es menor que el nodo con el que coincide.
- **a-star-search.** Realiza una búsqueda en grafo utilizando como estrategia la búsqueda A*.

Ejercicio 9:

Ver el camino seguido y la secuencia de acciones

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    BEGIN Exercise 9: Solution path / action sequence
;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Obtiene una lista de estados (nombres de planetas) desde
;; el nodo raíz hasta el nodo objetivo.
;;
;;  Input:
;;    node: nodo objetivo.
;;
;;  Returns:
;;    Lista de nombres que representa el camino desde
;;    el nodo raíz hasta el nodo objetivo.
(defun solution-path (node)
  (reverse (get-solution-path node)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Obtiene una lista de estados (nombres de planetas) desde
;; el nodo objetivo hasta el nodo raíz.
;;
;;  Input:
;;    node: nodo objetivo.
;;
;;  Returns:
;;    Lista de nombres que representa el camino desde
;;    el nodo objetivo hasta el nodo raíz.
(defun get-solution-path (node)
  (if (null node)
      nil
    (cons (node-state node) (get-solution-path (node-parent node)))))

;;;
;;; EJEMPLOS
;;;
(solution-path nil) ;;; -> NIL
(solution-path (a-star-search *galaxy-M35*))  ;;;-> (MALLORY ...)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Obtiene una lista de acciones desde el nodo raíz hasta el nodo objetivo.
;;
;;  Input:
;;    node: nodo objetivo.
;;
;;  Returns:
;;    Lista de nombres que representa las acciones desde
;;    el nodo raíz hasta el nodo objetivo.
(defun action-sequence (node)
  (reverse (get-action-sequence node)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Obtiene una lista de acciones desde el nodo objetivo hasta el nodo raíz.
;;
;;  Input:
;;    node: nodo objetivo.
;;
```

```
;; Returns:
;;   Lista de nombres que representa las acciones desde
;;   el nodo objetivo hasta el nodo raíz.
(defun get-action-sequence (node)
 (if (null node)
    nil
  (if (null (node-parent node))
     (node-action node)
   (cons (node-action node) (get-action-sequence (node-parent node))))))

;;;
;;;   END Exercise 9: Solution path / action sequence
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Comentario

El código solicitado por el ejercicio debe construir una lista de estados y una lista de acciones que representen la solución. Es decir, el camino seguido desde el nodo raíz hasta el nodo objetivo, pasando, si se ha indicado previamente, por los planetas obligatorios.

Para obtener la lista de nodos hemos implementado dos funciones:

- **get-solution-path.** Obtiene, de forma recursiva, una lista de estados desde el nodo objetivo hasta el nodo padre. Como el orden está invertido, necesitamos implementar un método adicional.
- **solution-path.** Obtiene la lista de estados desde el nodo objetivo al nodo padre e invierte su orden mediante una llamada a la función *reverse*. Así, devuelve la solución final.

Por otro lado, para obtener la lista de acciones, los dos métodos desarrollados son muy similares.

- **get-action-sequence.** Obtiene de manera recursiva la lista de acciones desde el nodo objetivo hasta el nodo padre.
- **action-sequence.** Invierte la lista que resulta de invocar a get-action-sequence y devuelve dicha lista como resultado final.

Ejercicio 10:
Otras estrategias de búsqueda

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   BEGIN Exercise 10: depth-first / breadth-first
;;;

;;
;; Función de búsqueda en profundidad
;;
;; Comprueba si la profundidad de node-1 es
;; mayor o igual que la de node-2.
;;
(defun depth-first-node-compare-p (node-1 node-2)
  t)
;;
;; Estrategia de búsqueda en profundidad
;;
(defparameter *depth-first*
  (make-strategy
   :name 'depth-first
   :node-compare-p #'depth-first-node-compare-p))

;;;
;;; EJEMPLOS
;;;
(solution-path (graph-search *galaxy-M35* *depth-first*))
(action-sequence (graph-search *galaxy-M35* *depth-first*))
;;; -> (MALLORY ... )

;;
;; Función de búsqueda en anchura
;;
;; Comprueba si la profundidad de node-1 es
;; menor o igual que la de node-2.
;;
(defun breadth-first-node-compare-p (node-1 node-2)
  nil)
;;
;; Estrategia de búsqueda en anchura
;;
(defparameter *breadth-first*
  (make-strategy
   :name 'breadth-first
   :node-compare-p #'breadth-first-node-compare-p))

;;;
;;;   END Exercise 10: depth-first / breadth-first
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Comentario

En este ejercicio hemos implementado las estrategias de búsqueda en anchura y en profundidad como dos funciones que devuelven *nil* y *t*, respectivamente.

La primera devuelve *nil* porque en la búsqueda en achura nos interesa que los nodos generados al expandir el nodo padre se introduzcan al final de la lista abierta, ya que primero expandimos los nodos más antiguos o de profundidad menor.

La función de comparación de la búsqueda en profundidad devuelve *t* porque, al contrario que en la búsqueda en anchura, se exploran primero los nodos generados más recientemente (o los más profundos). Por tanto, interesa que los nodos recién expandidos se encuentren al principio de la lista abierta.

REFLEXIÓN

Ejercicio 11:

Ejercicios de reflexión

11.1. ¿Por qué se ha realizado este diseño para resolver el problema de búsqueda? ¿Qué ventajas aporta? ¿Por qué se han utilizado funciones lambda para especificar el test objetivo, la heurística y los operadores del problema?

Al reunir todos los aspectos del problema bajo una estructura, en primer lugar forzamos la modularización del mismo. Estos dos factores nos permiten una estructura común desde la que enfocar el problema variando cualquiera de sus características con un esfuerzo de implementación nulo de cara al problema en si (aunque, por ejemplo, nuevas heurísticas o galaxias distintas requieran de su implementación propia).

Por último, las funciones utilizadas son lambda dado que no se van a utilizar en otros ámbitos, logramos un código más transparente, así como ahorramos memoria en el espacio de nombres de funciones durante la ejecución de nuestro código.

11.2. Sabiendo que en cada nodo de búsqueda hay un campo "parent", que proporciona una referencia al nodo a partir del cual se ha generado el actual ¿es eficiente el uso de memoria?

Por supuesto; al implementarse mediante referencia, en vez de repetir redundantemente la información en distintos espacios de memoria, todos referencian a un mismo espacio común en el que se encuentra la información de cada nodo, de forma que podemos generar estructuras relativamente grandes con un uso de memoria relativamente bajo.

11.3 ¿Cuál es la complejidad espacial del algoritmo implementado?

La complejidad espacial (o uso de memoria) tiene, como caso base, el del algoritmo A*, alrededor de la cual ha girado, fundamentalmente, el desarrollo de la práctica, puede ser:

- **Subexponencial** en el caso de que $|\mathbf{h(n) - h^*(n)}| \leq O(\log \mathbf{h^*(n)})$.
- **Exponencial** en otro caso o en caso de **h albitrario**:

$$O(b^d)$$

Siendo **d** la **división** del **coste óptimo** entre el **coste mínimo por acción**.

En estos cálculos hay que tener en cuenta que en el caso de la implementación del problema desarrollada a lo largo de esta práctica, **n** no sólo son cada uno de los nodos como los estados en si, si no como cada uno de estos repetido en base a la cantidad de estadios de completitud (osease, de nodos objetivo visitados) a lo largo de nuestro problema.

Osease, **respecto a un n básico** en un problema de encontrar un camino del nodo inicial al nodo objetivo, en nuestro planteamiento del problema habría que considerar un **n** igual al **n básico** multiplicado por todos los posibles estados de completitud del problema.

11.4 ¿Cuál es la complejidad temporal del algoritmo?

La complejidad espacial es equivalente a la temporal en los algoritmos A*, así que podemos tomar las mismas consideraciones expuestas en el **ejercicio 11.3**.

11.5 Indicad qué partes del código se modificarían para limitar el número de veces que se puede utilizar la acción "navegar por agujeros de gusano" (bidireccionales).

Nos cuesta entender lo que se solicita con este ejercicio.

La limitación de la función navigate ya está implementada en el código del ejercicio 8, no incluyendo nodos equivalentes a los presentes en la lista cerrada para continuar la ejecución de la función de navegación, lo cual ya limita la cantidad de veces que puede utilizar la acción "navegar por agujeros de gusano", impidiendo que la ejecución de código caiga en ningún bucle infinito.

Sección concreta de código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; A partir de un problema de búsqueda, busca la solución
;; óptima (camino más corto), de forma recursiva, desde el planeta
;; de origen hasta el planeta de destino, siguiendo una estrategia
;; definida.
;;
;;  Input:
;;    open-nodes: lista abierta que contiene los nodos generados,
;;                pero no expandidos.
;;    closed-nodes: lista cerrada que contiene los nodos generados
;;                  y expandidos previamente.
;;    problem: problema de búsqueda.
;;    strategy: estrategia de búsqueda.
;;
;;  Returns:
;;    Camino desde el nodo raíz hasta el nodo objetivo.
(defun graph-search-rec (open-nodes closed-nodes problem strategy)
  (if (null open-nodes)
      nil
    (let ((current-node (first open-nodes)))
      ;Comprueba si el nodo a expandir es el objetivo.
      (if (f-goal-test-galaxy current-node *planets-destination* *planets-mandatory*)
          ;Si lo es, lo devuelve como solución.
          current-node
        ;En caso contrario, comprueba si el nodo no está en la lista cerrada o,
        ;si está en ella, si tiene un valor de g inferior al primer nodo de closed-nodes.
        (if (not-in-closed-nodes current-node closed-nodes)
            ;Expande el nodo actual e inserta los hijos en open-nodes, ordenados de
            ;acuerdo al criterio de comparación de strategy.
            ;También inserta el nodo actual en la lista cerrada closed-nodes.
            (let ((new-open-nodes (insert-nodes-strategy (expand-node current-node problem) open-nodes
strategy))
                  (new-closed-nodes (append (list current-node) closed-nodes)))
              ;Continúa la búsqueda eliminando el nodo expandido actual
              ;de la lista abierta.
              (graph-search-rec (remove current-node new-open-nodes) new-closed-nodes problem strategy))
          ;Si el nodo a expandir no cumple las condiciones, se elimina directamente
          ;de la lista abierta.
          (graph-search-rec (remove current-node open-nodes) closed-nodes problem strategy))))))
```